LZW compression is a lossless data compression algorithm that was invented by Abraham Lempel, Jacob Ziv, and Terry Welch to reduce a file into a smaller size. The LZW compression algorithm compresses a file into a smaller file using a symbol table lookup algorithm where the algorithm relies on reoccurring patterns to save data space. Nowadays, this algorithm is most commonly used in GIF images, and optionally in PDF and TIFF. The LZW compression algorithm is a simple and versatile program that not only helps many computers save data space, but also help technology evolve.

LZW compression works by relying heavily on a symbol table (also can be called a dictionary). This table is initialized with 256 ASCII characters (because each ASCII character has 8-bits) where the characters will be considered as the substrings for the uncompressed file. The algorithm will then iterate through the file where the program scans to find the longest prefix within the dictionary. For example, when looking at the string "thisisthe", take the current character, t (ASCII value: 116), and look at the next character, h (104), and check if "th" is in the dictionary. Because it is not within the dictionary, we output "t" and add substring "th" to the dictionary with a value of 256 (it does not exist within the dictionary). If "th" was already in the dictionary, then we would look at the next character (in this case it's the word "the"), e, and add the word "the" to the dictionary, with its corresponding value. When expanding a compressed file, we need to know how many bits were used in compression. For example, if we want to expand the string, 116 104 105 115 258 256 101, we need to recreate the dictionary by looking at each value. Similar to compression, we look at the current value, 116, and the next value, 104, and check if 116 104 (which would contain the value of 256) is in our dictionary. Because it is not in the dictionary we add it to the dictionary and repeat this method as we iterate through the string until we get the expanded string, "thisisthe".

After explaining how the LZW compression algorithm works, different factors such as fixed/variable-length encoding can affect how well a file can be compressed. For example, the original LZW.java file uses fixed-length encoding where each codeword has a length of 12 bits. When looking at LZW.java, this means that the size of the codebook never exceeds 2^{12} and each codeword written is 12 bits. I wanted to improve on this using variable-length encoding where the minimum bit length for added codewords to the dictionary is 9 and the maximum bit length is 16. This means that the codebook will start small, allowing for smaller bits (and not take up unnecessary space as with the original LZW.java), and be able to grow to hold larger codewords for better compression. To implement this change, I took out the word "final" for the codewordBitlen (bit length) and codewordSize (size of the dictionary) from the LZW.java so that as the dictionary fills up, the codewordBitlen will increment by one starting at 8 until it reaches the MAX_CODEWORD_BITLEN (16). Because we allowed codewordBitlen to change, the size of the codebook, codewordSize, also increases at $2^{\text{codewordBitlen}}$ as the bit length increases.

In addition to changing it from fixed-length to variable-length, I also added options for the user on what to do when the dictionary gets full: a do nothing (n), reset (r), and monitor mode (m). Do nothing mode, does as it implies, when the dictionary is completely filled, do nothing and continue to use the full codebook. In reset mode, once the dictionary is filled (codewordBitlen = 2^{16}), the dictionary will be reset to try to compress the file further by removing the more common words in the file, which would lessen the compression. In monitor mode, the file would compress normally until dictionary was filled. Once the dictionary was filled it would take the old compression ratio (before the dictionary was filled) and divide it by the new compression ratio (after it was filled). If this ratio degrades by more than a set threshold (1.1), then the dictionary is reset. To make these modes work, I grouped the modes using an

enumerator called CompressMode.java, where each mode's method would be grouped with that mode as well as created a buildCompressionDictTable method. This method is useful for reset and monitor mode where once the dictionary table is full, when talking about reset mode, it will create a new table and reset the all the values (bit length, dictionary size, and codewords). In addition, I made a method to keep track of the ratio in monitor mode (after the dictionary is full) such that if the ratio is greater than 1.1 then it will reset the dictionary table. When calculating amount of data both read and written for the longest substring I multiplied its length by 16 which was added onto the uncompressed file sizes and incremented the bit length.

Similar to compression, because the compression method reads the first character to determine the mode used, expansion also must read the first character to determine what mode the file is in. The expansion method has been changed similarly to the compress method in that it keeps track of the current codeword value (i) and that for every value read, j, from the uncompressed file it finds the codeword corresponding to the value in the dictionary[j] and writes it to the uncompressed file. The expansion method works by finding the string in the dictionary and add it to the previous character added to the results. When a file is compressed using reset mode, if it is expanded, it will create a string array, initialize all 256 ASCII characters, set the bit length to 8 as well as the dictionary size to be 2^8 and add until it is full. If the expand method encounters the monitor method, then it will calculate the old and new compression ratio in the same way that the compress method does, but instead will add the first character of the codeword to previous codeword. It also resizes if the dictionary becomes full,in both reset and monitor mode, with its own buildExpansionDictTable() method.

Although there were not many changes with MyLZW.java and not drastic differences. Some issues could be that compression could drop several bytes of data which could invalidate

some data when expanding it (invalidating it being lossless). However, these changes to LZW.java helped the dictionary adapt better using variable-length encoding as well as demonstrate options for the user on which modes worked better for which files when expanding and compressing, where the goal is for the file to be the smallest it can be without losing any data.